7g rapport

Henrik Flindt
Nicolas Dyhrman
Adrian Joensen

November 21, 2018

1 Manual

Awari (or as it is called in its original tongue Oware) is a game in the Mancala family of board games played through out the world. To play the game by double clicking playAvari.exe. Player 1 goes first. The player must select a row between 1-6 that contain more than zero beans. For more rules about the game see https://en.wikipedia.org/wiki/Oware

2 Design

As part of the game development assigment it war required that the programming paradigm "Functional programming" was used. To ensure that the distribution of the beans on the board would be consistent and easily transformed between different functions it was decided to keep the board as an abstract form in a list. Visually, it was decided a command prompt (CMD) as the user interface insted of developing an actual graphic user interface.

3 Implementation

As part of the assigment a libary was given. While some of these functions were directly implemented as intented, some were modified, or removed. Through out the source code there are several print statments that are commented out. These are left deliberately as they serves as a great aid in debugging. A few extra functions were also implemented when required. While there are 13 functions in all, only those deemed non-trivial is explained in its full.
To see the full source code, see last part of this rapport. A brief overview is show below in their occurring order.
clearPit (l: int list) p Clears the element that corresponds to the chosen pit in the list.
matchOppsitePit (p: pit): pit
Returns the opposite pit number as in realation to the indexes of the board

list. To ensure that the compiler would compile with no warnings, an default function was added.

emptyPit (b: board) (p: pit) : board * pit

The most technically advanced move in Awari is the zero-steal move. If a bean lands in a empty pit the `distribute` calls this function. Depending upon what player makes the zero-steal move, the function creates several small lists containing the pits values before and after the pit p containing zero, and the opposition pit op (This value is returned from `matchOppsitePit`). Finally all these small list are truncated and returned.

rec distribute (l: board) (p : pit) (b : int) (player : player) : board * pit

Creates a new board list where the pit selected by the player is set to zero, and based upon how many beans it did contain increments the subsequent pits with 1. It a important if check to ensure that a hit to an empty pit will trigger the zero-steal move. It also checks that the move is not done to the players home pit!

If the p value passes the end of the list, distribute is recursively called to let the beans be distrusted around the board.

printBoard(b: board): unit

Prints the board. Several minor charchters are added to aid the understanding og how the game works. Each pit is separated by "|", an arrow shows the way the game is played, and "Awari" for style. Due to the way the print function works combined with how the list is oriented in the board, the spacial locality principle is violated, but this is not considered a problem with such a short list.

findWinner (b : board) : string

Takes the elements that correspond to the players' homes and checks who had won, or an potential draw. It is at the time of writing not been possible to play a "natural" game resulting in a draw.

isGameOver (b : board) : bool

Returns true if either side is empty. (i.e if board list [1..6] or [8..13] has the sum zero)

isHome (b : board) (p : player) (i : pit) : bool

Checks if the player is on his home pit.

pitEmpty (b:board)(i:int)(x:pit): pit

Checks if a selected pit is empty, by checking the sum of board list at that index is zero. If so returns a -1.

reverseNumbers(i:int): int

Takes a number given by the user and revers its to match the internal list.

getMove (b : board) (p:player) (q:string) : pit

Receives the players' input from the CMD and check if it is a legit move. (i.e. not empty and withing the range of 1-6)

turn (b : board) (p : player) : board

Checks controls that the turn changes between player one and two, and calls the repeat function if the move is illegal.

rec repeat (b: board) (p: player) (n: int) (t : bool) : board

Makes a large amount of calls to ensure that the game is over and if the moves

being done is legal.
play:
The main function that gets passed the staring board. This function is allows a to make large effective tests of all the parts at once.

4 White Box Testing

The whitebox testing was done eighter by testing specific functions isolated, or by calling `play` but with special case board to see how the code reacted to specific inputs. (4 different win conditions and two draw conditions.) The different outputs are shown with each test here. **clearPit test**
`clearPit` is checked if it empties the element of the list if index 1, 3, 7 or 13 is given.

```
C:\Users\Benadikt\Desktop\rapport\pop\7g>fsharpi awariLibIncomplete.fs clearPitWhiteBox.fsx
pit 1 now haves value 0 was this correct?result list
[0; 0; 3; 3; 3; 3; 3; 0; 3; 3; 3; 3; 3; 3] expected value
[0; 0; 3; 3; 3; 3; 3; 0; 3; 3; 3; 3; 3; 3] true


pit 3 now haves value 0 was this correct?result list
[0; 3; 3; 0; 3; 3; 3; 0; 3; 3; 3; 3; 3; 3] expected value
[0; 3; 3; 0; 3; 3; 3; 0; 3; 3; 3; 3; 3; 3] true


pit 7 now haves value 0 was this correct?result list
[0; 3; 3; 3; 3; 3; 3; 0; 3; 3; 3; 3; 3; 3] expected value
[0; 3; 3; 3; 3; 3; 3; 0; 3; 3; 3; 3; 3; 3] true


pit 13 now haves value 0 was this correct?result list
[0; 3; 3; 3; 3; 3; 3; 0; 3; 3; 3; 3; 3; 0] expected value
[0; 3; 3; 3; 3; 3; 3; 0; 3; 3; 3; 3; 3; 0] true
```

distribute check

For `distribute` it was check if the function did the arithmetic correct remaining beans (bolds left) from the right place(pit)

```
C:\Users\Benadikt\Desktop\rapport\pop\7g>fsharpi awariLibIncomplete.fs distributeWhiteBox.fsx
pit: 1
 bolds left: 1
pit: 1 is function output as expected ([0; 4; 3; 3; 3; 3; 3; 0; 3; 3; 3; 3; 3; 3], 1) true
pit: 13
 bolds left: 1
pit: 13 is function output as expected ([0; 3; 3; 3; 3; 3; 3; 0; 3; 3; 3; 3; 3; 4], 13) true
pit: 13
 bolds left: 2
pit: 0
 bolds left: 1
pit: 13 is function output as expected ([1; 3; 3; 3; 3; 3; 3; 0; 3; 3; 3; 3; 3; 4], 0) true
pit: 1
 bolds left: 3
pit: 2
 bolds left: 2
pit: 3
 bolds left: 1
pit: 1 is function output as expected ([0; 4; 4; 4; 3; 3; 3; 0; 3; 3; 3; 3; 3; 3], 3) true
```

win condition check

Here a boolean return is checks if it is the correct winner.

```
C:\Users\Benadikt\Desktop\rapport\pop\7g>fsharpi awariLibIncomplete.fs findWinnerWhiteBox.fsx
player 1 won is expected is expected result the same true
player 2 won is expected is expected result the same true
```

game over check

A series of true-false checks of the `gameOver`

```
C:\Users\Benadikt\Desktop\rapport\pop\7g>fsharpi awariLibIncomplete.fs isGameOverWhiteBox.fsx
is game over true is this same as expected? true
is game over false is this same as expected? true
is game over false is this same as expected? true
is game over true is this same as expected? true
is game over true is this same as expected? true
```

isHome check

A series of checks going through the different output of `isHome` function.

```
C:\Users\Benadikt\Desktop\rapport\pop\7g>fsharpi awariLibIncomplete.fs isHomeWhiteBox.fsx
player one with empty board result false is it as expected           true
player one where is in home result true is it as expected           true
player one where he is in pit 3 result false is it as expected           true
player one where he is in opposite home result false is it as expected           true
player two with empty board result false is it as expected           true
player two while being home true is it as expected           true
player two on pit 10 (on own side) result false is it as expected           true
player two while being in opposite home result false is it as expected           true
```

getMove check

Differet checks based upon user input and if they return the correct response.

```
C:\sources\github\pop\7g>mono getMoveWhitebox.exe
set move
3
getmove on with input 3 returns 4 is this same as expected true
set move
0
getmove on with input 0 returns -1 is this same as expected true
set move
7
getmove on with input 7 returns -1 is this same as expected true
set move
abc
getmove on with input abc returns -1 is this same as expected true
```

emptyPit check

Checks that the function `emptyPit` returns correct if the players land on an empty pit.

```
C:\Users\Benadikt\Desktop\rapport\pop\7g>fsharpi awariLibIncomplete.fs emptyPitWhiteBox.fsx
Enters matchOppsitePit
player 1 hit empty field and result ([0; 0; 3; 0; 3; 3; 0; 4; 3; 3; 3; 3; 3; 0], 1) was this as expected true
Enters matchOppsitePit
player 1 hit empty field and result ([0; 0; 3; 0; 0; 3; 0; 7; 3; 3; 0; 3; 3; 3], 4) was this as expected true
Enters matchOppsitePit
player 1 hit empty field and result ([0; 0; 3; 0; 3; 3; 0; 4; 0; 3; 3; 3; 3; 3], 6) was this as expected true
Enters matchOppsitePit
player 2 hit empty field and result ([4; 0; 3; 3; 3; 3; 3; 0; 0; 3; 0; 3; 3; 0], 8) was this as expected true
Enters matchOppsitePit
player 2 hit empty field and result ([4; 0; 3; 3; 3; 0; 3; 3; 8; 9; 0; 3; 3; 0], 10) was this as expected true
```

reverseNumber check

Checks if the correct reversed numbers are returned.

```
C:\Users\Benadikt\Desktop\rapport\pop\7g>fsharpi awariLibIncomplete.fs reverseNumbersWhiteBox.fsx
With input 1 we get 6 are this as expected true
With input 3 we get 4 are this as expected true
With input 6 we get 1 are this as expected true
With input 7 we get -1 are this as expected true
```

pitEmpty check

second last it was checked if a empty pit can be selected.

```
C:\Users\Benadikt\Desktop\rapport\pop\7g>fsharpi awariLibIncomplete.fs pitEmptyWhiteBox.fsx
is the pit empty -1 and are expected result the same true
is the pit empty 2 and are expected result the same true
```

Large game test check

Finally a large test was done to check if most functions could pass arguments. This was done by passing different boards lists to the `play` and monitor outputs.

All tests passed the tests given in the end.

5 Black Box testing

While no requirements for black box testing was in the assignment, this was indirectly done by playing the game. No issues appeared upon beta testing.

6 Conclusion

Visually the screen was a little difficult to distinguish since the CMD is not an ideal interface for games, and the usage of a list that had a mismatch with the board and user input options, caused quit a large amount of grief. While developing a graphic user interface would have required more code and testing, the amount of work potentially saved debugging errors with a list that mismatch with the user input would potentially have been worth the effort. A large set of whitebox testing indicates that code functions as required. Potential bugs will most likely be fixed by the Awari modding community.

7 Source Code

```
module Awari
type pit = int
type board = int list
type player = Player1 | Player2


let clearPit (l: int list) p =
let a = l.[0..p-1]
let b = [0]
let c = l.[p+1..13]
a @ b @ c

let matchOppsitePit (p: pit): pit =
//printfn "Enters matchOppsitePit"
match p with
| 1 -> 13
| 2 -> 12
| 3 -> 11
| 4 -> 10
| 5 -> 9
| 6 -> 8
| 8 -> 6
| 9 -> 5
| 10 -> 4
| 11 -> 3
| 12 -> 2
| 13 -> 1
| _ -> -1

let emptyPit (b: board) (p: pit) (player: player): board * pit =
//printfn "Enters emptyPit"
if player = Player1 then
```

```
let op = matchOppsitePit p
//printfn "Exit matchOppesitePit"
let a = b.[..p-1]
let c = [0]
let d = b.[p+1..6]
let home = [b.[7]+b.[p]+b.[op]+1]
let f = b.[8..op-1]
let g = [0]
let h = b.[op+1..13]
let uL = a @ c @ d @ home @ f @ g @ h
(uL, p)
else
//printfn "pit id = %i" p
let op = matchOppsitePit p
//printfn "oPpit id = %i" op
//printfn "Exit matchOppesitePit"
let home = [b.[0]+b.[p]+b.[op]+1]
let a = b.[..(op-1)]
let c = [0]
let d = b.[op+1..7]
let e = b.[8..p-1]
let f = [0]
let g = b.[p+1..13]
//printfn "aLength= %i" a.Length
if a.Length < 2 then
let uL = home @ b.[1..2] @ c @ d @ e @ f @ g
//printfn "%A" uL
//printfn "a.Length %i" uL.Length
(uL, p)
else
let uL = home @ a.Tail @ c @ d @ e @ f @ g
//printfn "other.Length %i" uL.Length
(uL, p)

let rec distribute (l: board) (p : pit) (b : int) (player : player) : board * pit =
let a = l.[..p-1]
let d = [l.[p]+1]
let c = l.[p+1..] //if p = 0 && b = 1 => playerhome + zero pit + obs pit.
if ((l.[p]=0) && (b = 1) && (not (p = 0)) && (not (p = 7))) then
//printfn "pit: %i\n bolds left: %i" p b
emptyPit l p player
else
let uL = a @ d @ c

//printfn "pit: %i\n bolds left: %i" p b
```

```
if p >= 13 then
if (b <> 1) then
distribute uL 0 (b-1) player
else
(uL, p)
elif b <= 1 then
(uL, p)
else
distribute uL (p+1) (b-1) player

//printfn "pit: %i\n bolds left: %i" p b



let printBoard(b: board): unit = //For printing the board a variation of the Maurits-printi
printfn "\n    1   2   3   4   5   6\n           <--         \n    %i | %i | %i | %i | %i |
//Spacial locality? What is that...

let findWinner (b : board) : string =
let player1Pit = b.[7]
let player2Pit = b.[0]

if(player1Pit > player2Pit) then
sprintf "Player 1 won with %i points, while Player 2 had %i" player1Pit player2Pit
elif(player1Pit < player2Pit) then
sprintf "Player 2 won with %i points, while Player 1 had %i" player2Pit player1Pit
else sprintf "Both players are drawn with a score of %i:%i!"player1Pit player2Pit



let isGameOver (b : board) : bool =
if b.IsEmpty then
true
else
// checker om player1's side består af 0 pinde
let player2gameover = List.forall (fun elem -> elem = 0) b.[1..6]
// checker om player2's side består af 0 pinde
let player1gameover = List.forall (fun elem -> elem = 0) b.[8..13]

// Returner resultater fra udregninger ovenfor
if player1gameover then
player1gameover
elif player2gameover then
player2gameover
else
false
```

```
let isHome (b : board) (p : player) (i : pit) : bool =

// Hvis listen er tom er der ingen hjem derfor return false
if b.IsEmpty then
false
else
// Finder ud af hvor stort halvdelen af boardet er
let halfBoardLen = b.Length / 2

// Plyayer 1's hjem er det første elem (kan også udregnes som 0) men det samme som halvdeler
let player1Home = halfBoardLen - halfBoardLen

// Player 2's hjem kan udregnes ved at halvere boardets længde (kan også bare laves som hal
let player2Home = b.Length - halfBoardLen

// Checker hvilken spiller der skal tjekkes om er hjemme
match p with
| Player1 ->
if i = 7 then
true
else
false

| Player2 ->
if i = 0 then
true
else
false

let getOppositePit (bLen : int) (i : pit) (p : player) : pit =
match p with
| Player1     -> (bLen / 2) - abs i
| Player2     -> (bLen / 2) + abs i


// pit 1 = player 1's pit
// pit 2 = player 2's pit
let CreateNewBoardFromHitEmptyPit (board : board) (pit1 : pit) (pit2 : pit) (p : player) : b
printfn "Player 1"

match p with
| Player1 ->
let newHomeValue = board.[7] + board.[pit2] + 1
let a = board.[0 .. (pit1-1)]
let b = [0]
```

```
let c = board.[(pit1+1) .. 6]
let d = [newHomeValue]
let e = board.[8 .. (pit2-1)]
let f = [0]
let g = board.[(pit2+1) .. 13]
a @ b @ c @ d @ e @ f @ g

| Player2 ->
let newHomeValue = board.[0] + board.[pit1] + 1
let a = [newHomeValue]
let b = board.[1 .. (pit1-1)]
let c = [0]
let d = board.[(pit1+1) .. 6]
let e = board.[8 .. (pit2-1)]
let f = [0]
let g = board.[(pit2+1) .. 13]
a @ b @ c @ d @ e @ f @ g


let HitEmptyPit (b : board) (i : pit) (p : player) =
if (isHome b p i) then
b
else
let player1 = getOppositePit (b.Length) i Player1
let player2 = getOppositePit (b.Length) i Player2
let newBoard = CreateNewBoardFromHitEmptyPit b player1 player2 p
newBoard


//Checks if a pit is epmty. If so returns -1, else retuns the object pit.
let pitEmpty (b:board)(i:int)(x:pit): pit =
if ((b.Item(i))=0) then  -1 else x;

//Since the board list goes the reveresd of the numbers player one uses, the numbers are mat
let reverseNumbers(i:int): int =
match i with
| 1 -> 6
| 2 -> 5
| 3 -> 4
| 4 -> 3
| 5 -> 2
| 6 -> 1
| _ -> -1 //Added to make the compiler stop complaing about unmatched exceptions.

let getMove (b : board) (p:player) (q:string) : pit =
printfn "%s" q
```

```
let userInput = System.Console.ReadLine()

let userInt = System.Int32.TryParse(userInput)

match userInt with
| (true, pitValue) ->
if pitValue < 7 && pitValue > 0 then
match p with
| Player1     -> pitEmpty b (reverseNumbers(snd(userInt))) ((b.Length / 2) - abs pitValue)
| Player2     -> pitEmpty b (snd(userInt)+7) ((b.Length / 2) + abs pitValue)
else
-1
| _           -> -1



let turn (b : board) (p : player) : board =

let rec repeat (b: board) (p: player) (n: int) (t : bool) : board =
printBoard b
let str =
if n = 0 then
if t then
sprintf "Invalid user input, please select number between 1 - 6, and be sure that there is c
else
sprintf "Player %A's move? " p
else
if t then
sprintf "Invalid user input, please select number between 1 - 6, and be sure that there is c
else
sprintf "Again?"
let i = getMove b p str
if i <> -1 then  //If move is true enteres this loop.

if(i = 13) then //If play2 selects last index in board list.
let (newB, finalPit) = (distribute (clearPit b i) (0) ( b.[i]) p)
if not (isHome b p finalPit) || (isGameOver newB) then
newB
else
repeat newB p (n + 1) false
else
let (newB, finalPit) = (distribute (clearPit b i) (1+i) (b.[i]) p)
if not (isHome b p finalPit) || (isGameOver newB) then
newB
else
```

```
repeat newB p (n + 1) false
else
repeat b p n true //If move is false.
repeat b p 0 false

let rec play (b : board) (p : player) : board =
if isGameOver b then
printfn "%s" (findWinner b)
b
else
let newB = turn b p
let nextP =
if p = Player1 then
Player2
else
Player1
//printfn "Before recursive"
play newB nextP
```